# The dialog manager, part 5: Converting a non-modal dialog box to modal

devblogs.microsoft.com/oldnewthing/20050404-48

April 4, 2005

Raymond Chen

Let's apply what we learned from last time and convert a modeless dialog box into a modal one. As always, start with the scratch program and make the following additions:

```
INT_PTR CALLBACK DlgProc(
    HWND hdlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
 switch (uMsg) {
 case WM_INITDIALOG:
  SetWindowLongPtr(hdlg, DWLP_USER, lParam);
  return TRUE;
 case WM_COMMAND:
  switch (GET_WM_COMMAND_ID(wParam, lParam)) {
  case IDOK:
   EndDialog(hdlg, 2005);
   break;
  case IDCANCEL:
   EndDialog(hdlg, 1776);
   break;
  }
 }
 return FALSE;
}
int DoModal(HWND hwnd)
{
 return DialogBox(g_hinst, MAKEINTRESOURCE(1), hwnd, DlgProc);
}
void OnChar(HWND hwnd, TCHAR ch, int cRepeat)
{
 switch (ch) {
 case ' ': DoModal(hwnd); break;
 }
}
// Add to WndProc
   HANDLE_MSG(hwnd, WM_CHAR, OnChar);
// Resource file
1 DIALOGEX DISCARDABLE  32, 32, 200, 40
STYLE DS_MODALFRAME | DS_SHELLFONT | WS_POPUP |
      WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Sample"
FONT 8, "MS Shell Dlg"
BEGIN
 DEFPUSHBUTTON "OK",IDOK,20,20,50,14
 PUSHBUTTON "Cancel",IDCANCEL,74,20,50,14
END
```

Not a very exciting program, I grant you that. It just displays a dialog box and returns a value that depends on which button you pressed. The `DoModal` function uses the `DialogBox` function to do the real work.

Now let's convert the `DoModal` function so it implements the modal loop directly. Why? Just to see how it's done. In real life, of course, there would normally be no reason to undertake this exercise; the dialog box manager does a fine job.

First, we need to figure out where we're going to keep track of the flag we called <dialog still active> last time. We'll keep it in a structure that we hang off the dialog box's `DWLP_USER` window bytes. (I sort of planned ahead for this by having the `DlgProc` function stash the `lParam` into the `DWLP_USER` extra bytes when the dialog is initialized.)

```
// fEnded tells us if the dialog has been ended.
// When ended, iResult contains the result code.
typedef struct DIALOGSTATE {
 BOOL fEnded;
 int iResult;
} DIALOGSTATE;
void EndManualModalDialog(HWND hdlg, int iResult)
{
 DIALOGSTATE *pds = reinterpret_cast<DIALOGSTATE*>
     (GetWindowLongPtr(hdlg, DWLP_USER));
 if (pds) {
  pds->iResult = iResult;
  pds->fEnded = TRUE;
 }
}
```

The `EndManualModalDialog` takes the place of the `EndDialog` function: Instead of updating the dialog manager's internal "is the dialog finished?" flag, we update ours.

All we have to do to convert our `DlgProc` from one using the dialog manager's modal loop to our custom modal loop, then, is to change the calls to `EndDialog` to call our function instead.

```
INT_PTR CALLBACK DlgProc(
    HWND hdlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
 switch (uMsg) {
 case WM_INITDIALOG:
  SetWindowLongPtr(hdlg, DWLP_USER, lParam);
  return TRUE;
 case WM_COMMAND:
  switch (GET_WM_COMMAND_ID(wParam, lParam)) {
  case IDOK:
   EndManualModeDialog(hdlg, 2005);
   break;
  case IDCANCEL:
   EndManualModeDialog(hdlg, 1776);
   break;
  }
 }
 return FALSE;
}
```

All that's left is to write the custom dialog message loop.

```
int DoModal(HWND hwnd)
{
 DIALOGSTATE ds = { 0 };
 HWND hdlg = CreateDialogParam(g_hinst, MAKEINTRESOURCE(1),
             hwnd, DlgProc, reinterpret_cast<LPARAM>(&ds));
 if (!hdlg) {
  return -1;
 }
 EnableWindow(hwnd, FALSE);
 MSG msg;
 msg.message = WM_NULL; // anything that isn't WM_QUIT
 while (!ds.fEnded && GetMessage(&msg, NULL, 0, 0)) {
  if (!IsDialogMessage(hdlg, &msg)) {
   TranslateMessage(&msg);
   DispatchMessage(&msg);
  }
 }
 if (msg.message == WM_QUIT) {
  PostQuitMessage((int)msg.wParam);
 }
 EnableWindow(hwnd, TRUE);
 DestroyWindow(hdlg);
 return ds.iResult;
}
```

Most of this should make sense given what we've learned over the past few days.

We start by creating the dialog modelessly, passing a pointer to our dialog state as the creation parameter, which as we noted earlier, our dialog procedure squirrels away in the `DWLP_USER` window bytes for `EndManualModalDialog` to use.

Next we disable the owner window; this is done after creating the modeless dialog, observing the rules for enabling and disabling windows. We then fall into our message loop, which looks exactly like what we said it should look. All we did was substitute `!ds.fEnded` for the pseudocode <dialog still active>. After the modal loop is done, we continue with the standard bookkeeping: Re-posting any quit message, re-enabling the owner before destroying the dialog, then returning the result.

As you can see, the basics of modal dialogs are really not that exciting. But now that you have this basic framework, you can start tinkering with it.

First, however, your homework is to find a bug in the above code. It's rather subtle. Hint: Look closely at the interaction between `EndManualModalDialog` and the modal message loop.

Raymond Chen

**Follow**